

JCE530 U. S. PTO  
01/22/99

Java™ Card™ Runtime Environment (JCРЕ) Specification

## 2.1 Speciation

2110

Copyright © 1998 Sun Microsystems, Inc.

All rights reserved. Copyright in this document is

THE JOURNAL OF CLIMATE

**SUN MICROSYSTEMS, INC. (SUN)<sup>®</sup>** hereby grants to you, at no charge, a non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense) under SUN's intellectual property rights that are essential to practice the Java™ Card™ Realtime Environment (JCРЕ™) 1.1 Specification (the "Specification") to use the Specification for internal evaluation purposes only. Other than this limited license, you acquire no right, title, or interest in or to the Specification and you shall have no right to use the Specification for productive or commercial use.

NETTIE M. ELLIOTT

19a, duplication, or disclosure by the U.S. Government is subject to reimbursement of costs under

SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. SUN SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.

1142

San Antonio, Inc.  
P.O. San Antonio Road  
Palo Alto, CA 94303 USA  
610 940-4100

Java™ Card™ Runtime Environment (JCRE) 2.1 Specification

Contents

Preface.....

1. Introduction.....

2. Lifetime of the Java Card Virtual Machine.....

3. Java Card APIlet Lifetime.....

3.1 The Method `install`.....

3.2 The Method `select`.....

3.3 The Method `process`.....

3.4 The Method `deselect`.....

3.5 Power Loss and Reset.....

4. Transient Objects.....

4.1 Events That Clear Transient Objects.....

5. Selection.....

5.1 The Default Applet.....

5.2 SELECT Command Processing.....

5.3 Non-SELECT Command Processing.....

6. Applet Redirection and Object Sharing.....

6.1 Applet Firewall.....

6.1.1 Contexts and Context Switching.....

Java™ Card™ Runtime Environment (JCRC) 2.1 Specification

6.1.2 Object Ownership.....

6.1.3 Object Access.....

6.1.4 Firewall Protection.....

6.1.5 Static Fields and Methods.....

6.2 Object Access Across Contexts.....

6.2.1 JCRC Event Point Objects.....

6.2.2 Global Arrays.....

6.2.3 JCRC Privileges.....

6.2.4 Shareable Interfaces.....

6.2.5 Determining the Previous Context.....

6.2.6 Shareable Interface Details.....

6.2.7 Obtaining Shareable Interface Objects.....

6.2.8 Object Access Behavior.....

6.3 Transient Objects and Applet contexts.....

7. Transactions and Atomicity.....

7.1 Atomicity.....

7.2 Transactions.....

7.3 Transaction Duration.....

7.4 Nested Transactions.....

7.5 Tear or Reset Transaction Failure.....

7.6 Aborting a Transaction.....

7.6.1 Programmatic Abortion.....

7.6.2 Abortion by the JCRC.....

7.6.3 Cleanup Responsibilities of the JCRC.....

7.7 Transient Objects.....

7.8 Cache Capacity.....

8. API Topics.....

8.1 The APDU Class.....

8.1.1 T=0 specifies for on-going data transfers.....

## Java™ Card™ Runtime Environment (JCRE) 2.1 Specification

8.1.2 T=1 specifies for outgoing data transfers	8-3
8.2 The security and crypto packages	8-4
8.3 JCSystem Class	8-5
9. Virtual Machine Topics	
9.1 Resource Failures	9-1
10. Applet Installed	
10.1 The Installer	10-1
10.1.1 Installer Implementation	10-1
10.1.2 Installer AID	10-2
10.1.3 Installer APPIDs	10-2
10.1.4 Installer Behavior	10-2
10.1.5 Installer Privileges	10-3
10.2 The Newly Installed Applet	10-3
10.2.1 Initialization Parameters	10-3

## Java™ Card™ Runtime Environment (JCERU) 2.1 Specification

### Preface

*Java™ Card™ technology combines a portion of the Java programming language with a runtime environment optimized for smart cards and related, small memory embedded devices. The goal of Java Card technology is to bring many of the benefits of Java software programming to the resource-constrained world of smart cards. This document is a specification of the Java Card Runtime Environment (JCERU) 2.1. A vendor of a Java Card-enabled device provides an implementation of the JCERU. A JCERU implementation within the context of this specification refers to a vendor's implementation of the Java Card Virtual Machine (VM), the Java Card Application Programming Interface (API), or other component, based on the Java Card technology specification. A *Reference Implementation* is an implementation produced by Sun Microsystems, Inc. APIs written for the Java Card platform are referred to as Java Card APIs.*

### Who Should Use This Specification?

This specification is intended to assist JCERU implementers in creating an implementation, developing a specification to extend the Java Card technology specifications, or in creating an extension to the Java Card Runtime Environment (JCERU). This specification is also intended for Java Card applet developers who want a greater understanding of the Java Card technology specifications.

### Before You Read This Specification

Before reading this guide, you should be familiar with the Java programming language, the Java Card technology specification, and smart card technology. A good resource for becoming familiar with Java technology and Java Card technology is the Sun Microsystems, Inc. website, located at: <http://java.sun.com>.

### How This Specification Is Organized

Chapter 1, "The Scope and Responsibilities of the JCERU," gives an overview of the services required of a JCERU implementation.

Chapter 2, "Lifetime of the Virtual Machine," defines the lifetime of the Virtual Machine.

## Java™ Card™ Runtime Environment (JCRE) 2.1 Specification

**Chapter 3, "Applet Lifetime,"** defines the lifetime of an applet.

**Chapter 4, "Transient Objects,"** provides an overview of transient objects.

**Chapter 5, "Selection,"** describes how the JCRE handles applet selection.

**Chapter 6, "Applet Isolation and Object Sharing,"** describes applet isolation and object sharing.

**Chapter 7, "Transactions and Atomicity,"** provides an overview of atomicity during transactions.

**Chapter 8, "API Rights,"** describes API functionality required of JCRE but not completely specified in the *Java Card 2.1 API Specification*.

**Chapter 9, "Virtual Machine Topics,"** describes virtual machine specific.

**Chapter 10, "Applet Installer,"** provides an overview of the Applet Installer.

**Chapter 11, "API Constants,"** provides the numeric value of constants that are not specified in the *Java Card API 2.1 Specification*.

**Glossary** is a list of words and their definitions to assist you in using this book.

## Related Documents and Publications

References to various documents or products are made in this manual. You should have the following documents available:

- *Java Card 2.1 API Draft 7 Specification*, Sun Microsystems, Inc.
- *Java Card 2.0 Language Model and Virtual Machine Specification, October 13, 1997, Revision 1.0 Final*, Sun Microsystems, Inc.
- *Java Card API Developer's Guide*, Sun Microsystems, Inc.
- *The Java Language Specification* by James Gosling, Bill Joy, and Guy L. Steele, Addison-Wesley, 1996, ISBN 0-201-53491-1.
- *The Java Virtual Machine Specification (Java Series)* by Tim Lindholm and Frank Yellin, Addison-Wesley, 1996, ISBN 0-201-63453-X.
- *The Java Class Library: An Annotated Reference (Java Series)* by Patrick Chan and Rosanna Lee, Addison-Wesley, two volumes, ISBN: 0201310023 and 0201110011.
- **ISO 7816 Specification Part 1-6.**
- **EN 450 Integrated Circuit Card Specification for Payment Systems.**

## 1. Introduction

The Java Card Runtime Environment (JCRE) 2.1 contains the Java Card Virtual Machine (VM), the Java Card Application Programming Interface (API) classes (and industry-specific extensions), and support services.

This document, the **JCRE 2.1 Specification**, specifies the JCRE functionality required by the Java Card technology. Any implementation of Java Card technology shall provide this necessary behavior and environment.

## 2. Lifetime of the Java Card Virtual Machine

In a PC or workstation, the Java Virtual Machine runs as an operating system in process. When the OS process is terminated, the Java applications and their objects are automatically destroyed.

In Java Card technology the execution lifetime of the Virtual Machine (VM) is the lifetime of the card. Most of the information stored on a card shall be preserved even when power is removed from the card. Persistent memory technology (such as EEPROM) enables a smart card to store information when power is removed. Since the VM and the objects it creates are used to represent application information that is persistent, the Java Card VM appears to run forever. When power is removed, the VM stops only temporarily. When the card is next read, the VM starts up again and recovers its previous object heap from persistent storage.

Aside from its persistent nature, the Java Card Virtual Machine is just like the Java Virtual Machine.

The card initialization time is the time after masking, and prior to the time of card personalization and issuance. At the time of card initialization, the JCERB is initialized. The framework objects created by the JCERB exist for the lifetime of the Virtual Machine. Because the execution lifetime of the Virtual Machine and the JCРЕ framework span CAD sessions of the card, the lifetimes of objects created by applets will also span CAD sessions. (CAD means Card Application Device, or card reader. Card sessions are those periods when the card is inserted in the CAD, powered up, and exchanging streams of ATRs with the CAD.) Objects that have this property are called persistent objects.

The JCРЕ implementer shall write an object persistence when:

- The Applet .register method is called. The JCРЕ stores a reference to the instance of the applet object. The JCРЕ implementer shall ensure that instances of class support are persistent.
- A reference to an object is stored in a field of any other persistent object or in a class's static field. This requirement stems from the need to preserve the integrity of the JCРЕ's internal data structures.

### 3. Java Card Applet Lifetime

For the purposes of this Specification, a Java Card applet is an instance of a class present in an applet's class file as source code, or is loaded into card memory, linked, and otherwise prepared for execution. (For the remainder of this Specification, the term "applet" refers to an applet written for the Java Card platform.) Applets registered with the applet's `registerApplet` method shall implement the methods described in this section.

When the supplier is installed on the smart card, the smart card in the terminal is removed from the terminal and the terminal is reconnected to the network. The SCRE shall not call the supplier's contractor directly.

### 3.1 The Method `install`

When `IInstal11` is called, no objects of the applet exist. The main task of the `IInstal11` method when one appears is to create an instance of the `Applet` class, and to register the instance. All other objects that the applet will need during its lifetime can be created as is feasible. Any other preparations necessary for the applet to be selected and executed by a C/C++ class can be done as is feasible. The `Instal11` method obtains initialization parameters from the contents of the unescaping byte array parameter.

Typically, an applet creates various objects, initializes them with predefined values, sets some internal state variables, and calls the `Applet::register` method to specify the AID (applet Identifier as defined in ISO 7816-2) to be used to select. This installation is considered successful when the call to `fileApplet->realizer` method completes without an exception. The installation is deemed unsuccessful if the `Instal11` method does not call the `Applet::realizer` method, or if an exception is thrown from within the `Instal11` method prior to the `Applet::register` method being called, or if the `Applet::register` method throws an exception. If the installation is unsuccessful the C/C++ shall perform all cleanup which is required. That is, all persistent objects shall be returned to the state they had prior to calling the `IInstal11` method. If the installation is successful, the C/C++ can mark the applet as available for selection.

### 3.3 The Method process

All APDU's are received by the JCRC, which passes an instance of the APDU class to the process method of the currently selected applet.

---

**Note - A SELECT APDU might cause a change in the currently selected applet prior to the call to the process method.**

On normal return, the JCRC automatically appends the 9000 as the completion response SW to any data already sent by the applet.

At any time during process, the applet may throw an ISODException with an appropriate SW, in which case the JCRC catches the exception and returns the SW to the CAD.

If any other exception is thrown during process, the JCRC catches the exception and returns the status word 15070165, SW\_UNIMPLEMENTED to the CAD.

### 3.2 The Method select

Applies remain in a suspended state until they are explicitly selected. Selection occurs when the JCRE receives a SELECT API call in which the name data matches the AID of the apply. Selection causes an apply to become the currently selected apply.

Prior to calling SELECT, the JCRE shall deselect the previously selected apply. The JCRE indicates this to the apply by invoking the apply's *deselect* method.

The JCRE informs the apply of selection by invoking its *select* method.

one application may decide to use a sequence of several **SELECT** and **PUT** commands to implement a more complex process. If the application returns **true**, the actual **SELECT** APDU command is supplied to the applet in the **onGetPdu** method. If the application returns **false**, the actual **SELECT** APDU command is supplied to the applet in the **onGetPdu** method, as that the applet can evaluate the APDU content. The applet can implement all of the processes method, as that the applet can evaluate the APDU content. The applet can process the **SELECT** APDU command exactly like it processes any other APDU command. It can respond to the **SELECT** APDU with data (see the process method for details), or it can flag errors by throwing an **IllegalArgumentException** with the appropriate SVU (return code **bad4**). The SVU and optional response data are returned to the ADL.

If the **apple** devices to be selected, the IC2B will return an AFDU response status word of 150, SW\_APPLE, SELECTED,ATTEN to the CxD. Upon selection failure, like IC2E status is set to indicate that no apple is selected.

### 3.4 The Method deselect

When the JCRE receives a SELECT APDU command in which the name matches the AID of an applet, the JCRE calls the DESSELECT method of the currently selected applet. This allows the applet to perform any cleanup operations that may be required in order to allow some other applet to execute.

The Applet.eselect method shall return false when called during the deselect method. Exceptions thrown by the deselect method are caught by the JCRE, but the applet is deselected.

### 3.5 Power Loss and Reset

Power loss occurs when the card is withdrawn from the CAD or if there is some other mechanical or electrical failure. When power is reapplied to the card and on CardReset (written or read) the JCRE shall ensure that:

- Transfus data is reset to the default value.
- The transaction in progress, if any, when power was lost (or reset occurred) becomes implicitly deselected. (In this case the deselect method is not called.)
- The applet that was selected when power was lost (or reset occurred) becomes implicitly deselected. (In this case the deselect method is not called.)
- If the JCRE implements default applet selection (see paragraph 3.1) the default applet is selected as the currently selected applet, and that the default applet's select method is called. Otherwise, the JCRE sets its state to indicate that no applet is selected.

## 4.1 Events That Clear Transient Objects

Transient objects are used for maintaining states that shall be preserved across card reads. When a transient object is created, one of two events are specified that cause its fields to be cleared: **CLEAR\_ON\_RESET** or **CLEAR\_ON\_DISELECT**. Transient objects are used for maintaining states that shall be preserved across card reads. **CLEAR\_ON\_DISELECT** transient objects are used for maintaining states that must be preserved while an applet is selected, but not across applet activations or card reads.

Details of the two clear events are as follows:

**CLEAR\_ON\_RESET**—the object's fields are cleared when the card is reset. When a card is powered on, this also causes a card reset.

**Note**—It is not necessary to clear the fields of transient objects before power is removed from a card. However, it is necessary to guarantee that the previous contents of such fields cannot be recovered once power is lost.

**CLEAR\_ON\_DISELECT**—the object's fields are cleared whenever any applet is deselected. Because a card read implicitly deselects the currently selected applet, the fields of **CLEAR\_ON\_DISELECT** objects are also cleared by this same event specified for **CLEAR\_ON\_RESET**.

The currently selected applet is explicitly deselected (its **deselect** method is called) only when a **SELECT** command is processed. The currently selected applet is deselected and then the fields of all **CLEAR\_ON\_DISELECT** transient objects are cleared regardless of whether two **SELECT** commands:

- Fail to select an applet.
- Selects a different applet.
- Re-selects the same applet.

A transient object within the Java Card platform has the following required behavior:

- The stack
- Local variables
- A class static field
- A field in another existing object

Transient objects require objects that contain temporary (transient) data that need not be preserved across CAD sessions. Java Card does not support the **do** or **try...finally** construct. However, Java Card technology provides methods to create transient arrays with primitive components or references to objects.

The term "transient object" is a misnomer. It can be incorrectly interpreted to mean that the object itself is transient. However, only the contents of the fields of the object (except for the length field) have a transient nature. As with any other object in the Java programming language, transient objects within the Java Card platform exist as long as they are referenced from:

## 5.2 SELECT Command Processing

The SELECT APDU command is used to select an applet. Its behavior is:

1. The SELECT APDU is always processed by the JCRE (regardless of which, if any, applet is active).
2. The JCRE matches its internal table for a matching AID. The JCRE shall support selecting an applet where the full AID is present in the SELECT command.

JCRE implementations are free to enhance their JCRE to support other selection criterion. An example of this is selection via partial AID match as specified in ISO 7816-4. The specific requirements are as follows:

**Note -** An octet is indicated binary bit numbering as in ISO 7816. Most significant bit = 0. Least significant bit = 8. Least significant bit = 0.

- 1) Applet SELECT command uses CLA=0x00, INS=0x40.
- 2) Applet SELECT command uses "Selection by DF name". Therefore, PI=0x04.
- 3) Any other value of PI implies that is not an applet select. The APDU is processed by the currently selected applet.
- 4) JCRE shall support exact DF name (AID) selection i.e P2=400000 x 00. (0x4000 are don't care).
- 5) All other partial DF name SELECT option (b2,b1)'s are JCRE implementation dependent.
- 6) All file control information option codes (b4,b3) shall be supported by the JCRE and interpreted and processed by the applet.

### 5.1 The Default Applet

Normally, applets become selected only via a successful SELECT command. However, some smart card/CAD applications require that there be a default applet that is implicitly selected after every card read. The behavior is:

1. After card read (or power on, which is a form of read) the JCRE performs its initializations and checks to see if its internal table indicates that a particular applet is the default applet. If so, the JCRE makes this applet the currently selected applet, and the applet's select method is called. If the applet's select method throws an exception or returns false, then the JCRE sets its state to indicate that no applet is selected. (The applet's process method is not called during default applet selection because there is no SELECT APDU) When a default applet is selected at card read, it shall not require its process method to be called.
2. The JCRE ensures that the ATR has been sent and the card is now ready to accept APDU commands. If a default applet was successfully selected, then APDU commands can be sent directly to this applet. If a default applet was not selected, then only SELECT commands can be processed.
3. The mechanism for specifying a default applet is not defined in the Java Card API 2.1. It is a JCRE implementation detail and is left to the individual JCRE implementor.

Notes...

If there is no matching AID, the SELECT command is forwarded to the currently selected applet (if any) for processing as a normal applet APDU command.

If there is a matching AID and the SELECT command fails, the JCRAF always enters the state where no applet is selected.

If the matching AID is the same as the currently selected applet, the JCRAF will go through the process of de-selecting the applet and then selecting it. Reselection should fail, leaving the card in a state where no applet is selected.

### 5.3 Non-SELECT Command Processing

When a non-SELECT APDU is received and there is no currently selected applet, the JCRAF shall respond to the APDU with status code 0x099 (SW\_APPLET\_SELECT\_FAILED).

When a non-SELECT APDU is received and there is a currently selected applet, the JCRAF invokes the process method of the currently selected applet passing the APDU as a parameter. This causes a context switch from the JCRAF context to the currently selected applet's context. When the process method exits, the VM switches back to the JCRAF context. The JCRAF sends a response APDU and waits for the next command APDU.

Most method invocations in Java Card technology do not cause a context switch. Context switches only occur during invocation of and return from certain methods, as well as during exception calls from those methods (see 6.2.8).

During a context-switching method invocation, an additional piece of data, indicating the currently active content, is pushed onto the return stack. This content is restored when the method is exited. Further details of contexts and context switching are provided in later sections of this chapter.

## 6. Applet Isolation and Object Sharing

Any implementation of the JCRE shall support isolation of content and applets. Isolation means that one applet can not access the fields or objects of an applet in another content unless the other applet explicitly provides an interface for notes. The JCRE mechanisms for applet isolation and object sharing are detailed in the sections below.

### 6.1 Applet Firewall

The *applet firewall* within Java Card technology is runtime-enforced protection used to separate from the Java technology protection. The Java language protection still apply to Java Card applets. The Java language ensures that strong typing and protection attributes are enforced.

Applet firewalls are always enforced in the Java Card VM. They allow the VM to automatically perform additional security checks at runtime.

#### 6.1.1 Contexts and Context Switching

Firewalls essentially partition the Java Card platform's object space into *separate* protected object spaces called contexts. The firewall is the boundary between one content and another. The JCRE shall allocate and manage an *applet* memory for each applet that is installed on the card. (But see paragraph 6.1.2 below for a discussion of group contexts.)

In addition, the JCRE maintains its own *JCRE context*. This context is much like an applet context, but it has special system privileges to do it can perform operations that are denied to applet contexts.

At any point in time, there is only one active context within the VM. (This is called the *currently active content*.) All bytecode that access objects are checked by runtime against the currently active context in order to determine if the access is allowed. A `java.lang.SecurityException` is thrown when an access is disallowed.

When certain well-defined conditions are met during the execution of invoke-type bytecodes as described in paragraph 6.2.8, the VM performs a context switch. The previous content is pushed onto an internal VM stack, a new content becomes the currently active content, and the invoked method executes in this new content. Upon exit from that modified file, VM performs a returning content switch. The original content (or the caller of the method) is popped from the stack and is restored as the currently active content. Context switches can be nested. The maximum depth depends on the amount of VM stack space available.

#### 6.1.2 Object Ownership

When a new object is created, it is associated with the currently active content. But the object is owned by the applet instance within the currently active content when the object is instantiated. An object is owned by an applet instance, or by the JCRE.

#### 6.1.3 Object Access

In general, an object can only be accessed by its owning content, that is, when the owning content is the currently active content. The firewall prevents an object from being accessed by another applet in a different content.

In implementation terms, each time an object is accessed, the object's owner content is compared to the currently active content. If these do not match, the access is not performed and a `SecurityException` is thrown.

An object is accessed when one of the following bytecodes is executed using the object's reference:

- `getfield`, `putfield`, `invokespecial`, `invokeinterface`, `checkcast`, `instanceof`
- `new` refers to the various types of array bytecodes, such as `bacl`, `baclw`, etc.

This list includes any special or optimized forms of these bytecodes implemented in the Java Card VM, such as `getfield_a`, `putfield_a`, etc.

#### 6.1.4 Firewall Protection

The Java Card firewall provides protection against the most frequently anticipated security concern: developer mistakes and design oversights that might allow sensitive data to be "leaked" to an other applet. An applet may be able to obtain an object reference from a publicly accessible location, but if the object is owned by a different applet, the firewall ensures security.

The firewall also provides protection against inserted code. If incorrect code is loaded onto a card, the firewall still protects objects from being accessed by this code.

The Java Card API 2.1 specifies the basic minimum protection requirements of contexts and firewalls because these features shall be supported in ways that are *not transparent* to the supplier developer. Developers shall be aware of the behavior of objects, APIs, and exceptions related to the firewall.

JCRE implementers are free to implement additional security mechanisms beyond those of the supplier firewall, as long as these mechanisms are transparent to applets and do not change the externally visible operation of the VM.

### 6.1.5 Static Fields and Methods

It should also be noted that classes are not owned by contexts. There is no runtime context check that can be performed when a class static field is accessed. Neither is there a context switch when a static method is invoked. (Similarly, it is not expected to need a context switch to invoke static methods.)

Public static fields and public static methods are accessible from any context. Static methods execute in the same context as their caller.

Objects referenced in static fields are just regular objects. They are owned by whatever created them and standard firewall access rules apply. If it is necessary to share them across multiple applet contexts, then these objects need to be *Shareable Interface Objects* (SIOs). (See paragraph 6.2.4 below.)

Of course, the conventional Java technology protection rules still enforced for static fields and methods. In addition, when applets are installed, the installer utilizes that each attempt to link to an external static field or method is permitted. Initialization and synthesis rules likewise are beyond the scope of this specification.

#### 6.1.5.1 Optional static access checks

The JCRE may perform optional runtime checks that are redundant with the constraints enforced by a verifier. A Java Card VM may detect when code violates fundamental language restrictions, such as invoking a private method in another class, and report an otherwise silentable illegal operation.

### 6.2 Object Access Across Contexts

To enable applets to interact with each other and with the JCRE, some well-defined yet secure mechanisms are provided so one context can access an object belonging to another context.

These mechanisms are provided in the Java Card API 2.1 and are discussed in the following sections:

- JCRE Entry Point Objects
- Global Arrays
- JCRE Privileges
- Shareable Interfaces

### 6.2.1 JCRE Entry Point Objects

Secure computer systems shall have a way for user-privileged user processes (that are restricted to a subset of resources) to request system services performed by privileged "System" entities.

In the Java Card API 2.1, this is accomplished using JCRE Entry Point Objects. These are objects owned by the JCRE context, but they have been flagged as containing entry point methods.

The firewall protects these objects from access by applets. The entry point designation allows the methods of these objects to be invoked from any context. When that occurs, a context switch to the JCRE context is performed. These methods are the gateways through which applets request privilege JCRE system services.

There are two categories of JCRE Entry Point Objects:

#### Temporary JCRE Entry Point Objects

Like all JCRE Entry Point Objects, methods of temporary JCRE Entry Point Objects can be invoked from any applet context. However, references to these objects cannot be stored in class variables, instance variables, or any array component. The JCRE detects and catches attempts to store references to these objects as part of the firewall functionality to prevent unauthorized re-use.

The APP1 object and all JCRE-owned exception objects are examples of temporary JCRE Entry Point Objects.

#### Permanent JCRE Entry Point Objects

Like all JCRE Entry Point Objects, methods of permanent JCRE Entry Point Objects can be invoked from any applet context. Additionally, references to these objects can be stored and freely re-used. JCRE-owned APP1 instances are examples of permanent JCRE Entry Point Objects.

The JCRE is responsible for:

- Determining what privileged services are provided to applets.
- Defining classes containing the entry point methods for those services.
- Creating one or more object instances of these classes.
- Designating these instances as JCRE Entry Point Objects.
- Designating JCRE Entry Point Objects as temporary or permanent.
- Making references to those objects available to applets as needed.

Note — Only the methods of these objects are accessible through the firewall. The fields of these objects are still affected by the firewall and can only be accessed by the JCRE context.

Only the JCRE itself can designate Entry Point Objects and whether they are temporary or permanent. JCRE implementers are responsible for implementing the mechanism by which JCRE Entry Point Objects are designated and how they become temporary or permanent.

### 6.2.2 Global Arrays

The global nature of some objects requires that they be accessible from any applet context. The firewall would ordinarily prevent these objects from being used in a flexible manner. The Java Card VM allows an object to be designated as *global*.

All global arrays are temporary global array objects. These objects are owned by the JCRE context. Null can be accessed from any applet context. However, references to these objects cannot be stored in class variables.

## Java™ Card™ Runtime Environment (JCRE) 2.1 Specification

instance variables or array contents. The JCRE detects and restricts attempts to store references to the objects as part of the firewall functionality to prevent unauthorized re-use.

For added security, only arrays can be designated as global and only the JCRE itself can designate global arrays. Because applets cannot create them, no API methods are defined. JCRE implementers are responsible for implementing the mechanism by which global arrays are designated.

At the time of publication of this specification, the only global arrays required in the Java Card API 2.1 are the AID buffer and the byte array input parameter (bav or rav) to the applet's install method.

**Note**—Because of its global status, the API specifies that the AID buffer is cleared to zero whenever an applet is selected, before the JCRE accepts a new APDU command. This is to prevent an applet's potentially sensitive data from being "leaked" to another applet via the global AID buffer. The AID buffer can be accessed from a situated interface object context and is available for passing data across applet contexts. The applet is responsible for protecting secret data that may be accessed from the AID buffer.

### 6.2.3 JCRE Privileges

Because it is the "system" context, the JCRE context has a special privilege: It can invoke a method of any object on the card. For example, assume that object X is owned by applet A. Normally, only owner A can invoke the fields and methods of X. But the JCRE context is allowed to invoke any of the methods of X. During an invocation, a context switch occurs from the JCRE context to the applet context that owns X.

**Note**—The JCRE can access both *methods* and *fields* of X. Method access is the mechanism by which the JCRE enters an applet context. Although the JCRE could invoke any method through the firewall, it shall only invoke the *select*, *process*, *deactivate*, and *getShareableInterfaceObject* (see 6.2.7.1) methods defined in the Applet class.

The JCRE context is the currently active context when the VM begins running after a *readReset*. The JCRE context is the "top" context and is always either the currently active context or the previous context saved on the stack.

### 6.2.4 Shareable Interfaces

Shareable interfaces are a new feature in the Java Card API 2.1 to enable applet interoperation. A shareable interface defines a set of shared interface methods. These interface methods can be invoked from one applet context even if the object implementing them is owned by another applet context.

In this specification, an object implementing a shareable interface is called a *Shareable Interface Object* (SIO).

To the owning context, the SIO is a normal object whose fields and methods can be accessed. To any other context, the SIO is an instance of the shareable interface, and only the methods defined in the shareable interface are accessible. All other fields and methods of the SIO are protected by the firewall.

Shareable interfaces provide a secure mechanism for inter-applet communication, as follows:

1. To make an object available to another applet, applet A first defines a shareable interface, SI. A shareable interface extends the interface JavaCard, ICardObject. All methods defined in the shareable interface, SI, represent the services that applet A makes available to other applets.
2. Applet A then defines a class C that implements the shareable interface SI. Class C implements the methods defined in SI. C may also define other methods and fields, but these are protected by the applet firewall. Only the methods defined in SI are accessible to other applets.

## Java™ Card™ Runtime Environment (JCRC) 2.1 Specification

1. Applet A creates an object instance O of class C. O belongs to applet A, and the firewall allows A to access any of the fields and methods of O.
4. To access applet A's object O, applet B creates an object reference SIO of type SI.
5. Applet B invokes a special method *getShareableInterfaceObject*, described in paragraph 6.7.2 to query a shareable interface object reference from applet A.
6. Applet A receives the request and the AID of the requester (B) via *Applet.setShareableInterfaceObject*, and determines whether or not it will share object O with applet B.
7. If applet A agrees to share with applet B, A responds to the request with a reference to O. This reference is cast to type Shareable so that none of the fields or methods of O are visible.
8. Applet B receives the object reference from applet A, casts it to type SI, and stores it in object reference SIO. Even though SIO actually refers to A's object O, SIO is of type SI. Only the shareable interface methods defined in SI are visible to B. The firewall prevents the other fields and methods of O from being accessed by B.
9. Applet B can request services from applet A by invoking one of the shareable interface methods of SIO. During the invocation the Java Card VM performs a context switch. The original currently active context (D) is saved on a stack and the context of the owner (A) of the actual object (O) becomes the new currently active context. A's implementation of the shareable interface method (SI) executes in A's domain, service for applet B.
10. The SI method can find out the AID of its owner (B) via the *getSystem.getPreviousContextAID* method. This is described in paragraph 6.2.5. The method determines whether or not it will perform the context switch. A's implementation of the shareable interface method (SI) executes in A's domain.
11. Because of the context switch, the firewall allows the SI method to access all the fields and methods of object O and any other object owned by B. At the same time, the firewall protects the method from accessing non-shared objects owned by B.
12. The SI method can access the parameters passed by B and can provide a return value to B.
13. During the return, the Java Card VM performs a restoring context switch. The original currently active context (D) is restored from the stack, and again becomes the current context.
14. Because of the context switch, the firewall again allows B to access any of its objects and previous D from accessing non-shared objects owned by A.

### 6.2.5 Determining the Previous Context

When an applet calls *getSystem.getPreviousContextAID*, the JCRE shall return the instance AID of the applet instance active at the time of the last context switch.

The JCRE context does not have an AID. If an applet calls the *getPreviousContextAID* method when the applet context was entered directly from the JCRE context, this method returns 0.

If the applet calls *getPreviousContextAID* from a method that may be accessed either from within the applet itself or when accessed via a shareable interface from an external applet, it shall check for null return before performing caller AID authentication.

## 6.2.6 Shareable Interface Details

A shareable interface is simply one that extends (either directly or indirectly) the `Tagging Interface`. JavaCard® `Shareable`™. This shareable interface is similar in concept to the `Remote Interface` used by the RMI facility, in which calls to the interface methods take place across a local/remote boundary.

### 6.2.6.1 The Java Card Shareable Interface

Java classes extending the `Shareable` or `Tagging` interface have this special property: calls to the interface methods take place across Java Card's applet boundary via a `Context Switch`.

The `Shareable` interface serves to identify all shared objects. Any object that needs to be shared through the applet firewall shall directly or indirectly implement this interface. Only those methods specified in a shareable interface are available through the firewall.

Implementation classes can implement any number of shareable interfaces and can extend other shareable implementation classes.

Like any Java plain interface, a shareable interface simply defines a set of service methods. A service provider class *implements* the shareable interface and provides implementations for each of the service methods of the interface. A service client class accesses the service by obtaining an object reference, casting it to the shareable interface type if necessary, and invoking the service methods of the interface, casting it to the shareable interface type if necessary, and invoking the service methods of the interface.

The `shareable` interface within the Java Card technology shall have the following properties:

- When a method in a shareable interface is invoked, a context switch occurs to the context of the object's owner.
- When the enclosed `exit`, the context of the caller is restored.
- Exception handling is enhanced so that the currently active context is *correctly restored* during the stack frame unwinding that occurs as an exception is thrown.

## 6.2.7 Obtaining Shareable Interface Objects

Inter-applet communication is accomplished when a client applet invokes a shareable interface method of a SIO belonging to a server applet. In order for this to work, there must be a way for the client applet to obtain the SIO from the server applet in the first place. The JCER provides a mechanism to make this possible. The `Applet class` and the `ObjectAccess class` provide methods to enable a client to request services from the server.

### 6.2.7.1 The Method `Applet.getShareableInterfaceObject`

This method is implemented by the server applet instance. It shall be called by the JCER to mediate between a client applet that requests to use an object belonging to another applet, and the server applet that makes its objects available for sharing.

The default behavior shall return null, which indicates that an applet does not participate in inter-applet communication.

A server applet that is intended to be invoked from another applet needs to override this method. This method should examine the `clientApplet` and the `parameter`. If the `clientApplet` is not one of the expected AIDs, the method should return null. Similarly, if the `parameter` is not recognized or is not allowed for the method, the method should return null.

### 6.2.7.2 The Method `JCSysystem.getAppletShareableInterfaceObject`

The `JCSysystem` class contains the method `getAppletShareableInterfaceObject`, which is invoked by a client applet to communicate with a server applet.

The JCER shall implement this method to behave as follows:

1. The JCER searches its internal applet table for one with `serverAID`. If not found, null is returned.
2. The JCER invokes this applet's `getShareableInterfaceObject` method, passing the `c1` parameter of the caller and the `parameter`.
  - This context switch occurs to the server applet, and its implementation of `getShareableInterfaceObject` proceeds as described in the previous section. The server applet returns a SIO (or null).
  - 4. `getAppletShareableInterfaceObject` returns the same SIO (or null) to its caller.
3. For enhanced security, the implementation shall make it impossible for the client to tell which of the following conditions caused a null value to be returned:
  - The serverAID was not found.
  - The server applet does not participate in inter-applet communication.
  - The server applet won't communicate with this client.
  - The server applet won't communicate with this client as specified by the parameter.

## 6.2.8 Class and Object Access Behavior

A static class field is declared when one of the following Java bytecode is executed:

`getSIO, putSIO, putStatic`

An object is accessed when one of the following Java bytecode is executed using the object's reference:

`getfield, putfield, invokevirtual, invokevirtual, athrow,`

`newarray, newstore, arraylength, checkcast, instanceof`

`new` refers to the various types of array bytecode, such as `ba1oad, sa1store, etc.`

This list also includes any special or optimized forms of these bytecode that may be implemented in the Java Card VM, such as `getfield, putfield, invokevirtual, invokevirtual, instanceof, etc.`

Prior to performing the work of the bytecode as specified by the Java VM, the Java Card VM will perform an access check on the referenced object. If access is denied, then a `SecurityException` is thrown.

The access checks performed by the Java Card VM depend on the type and owner of the referenced object, the bytecode, and the currently active context. They are described in the following sections.

### 6.2.8.1 Accessing Static Class Fields

**Bycodes:**

`getstatic`, `putstatic`

- If the JCRE is the currently active context, then access is allowed.
- Otherwise, if the bytecode is `putstatic` and the field being stored is a reference type and the reference being stored is a reference to a temporary JCRE Entry Point Object or a global array then access is denied.
- Otherwise, access is allowed.

### 6.2.8.2 Accessing Array Objects

**Bycodes:**

`<V>load`, `<T>store`, `arraylength`, `checkcast`, `instanceof`

- If the JCRE is the currently active context, then access is allowed.
- Otherwise, if the bytecode is `arraylength`, `checkcast`, or `instanceof` then access is denied.
- If the JCRE is the currently active context, then access is allowed.
- Otherwise, if the bytecode is `<V>load` or `<T>store` and the component being stored is a reference type and the reference being stored is a reference to a temporary JCRE Entry Point Object or a global array then access is denied.
- Otherwise, if the array is owned by the currently active context, then access is allowed.
- Otherwise, if the array is designated global, then access is allowed.
- Otherwise, access is denied.

### 6.2.8.3 Accessing Class Instance Object Fields

**Bycodes:**

`getfield`, `putfield`

- If the JCRE is the currently active context, then access is allowed.
- Otherwise, if the bytecode is `getfield` and the field being stored is a reference type and the reference being stored is a reference to a temporary JCRE Entry Point Object or a global array then access is denied.
- Otherwise, if the object is owned by the currently active context, then access is allowed.
- Otherwise, access is denied.

### 6.2.8.4 Accessing Class Instance Object Methods

**Bycodes:**

`invokevirtual`

- If the object is owned by the currently active context, then access is allowed. Context is switched to the object owner's context.
- Otherwise, if the object is designated a JCRE Entry Point Object, then access is allowed. Context is switched to the object owner's context.
- Otherwise, access is denied.

### 6.2.8.5 Accessing Standard Interface Methods

**Bycodes:**

`invokesuper`

- If the object is owned by the currently active context, then access is allowed.
- Otherwise, if the JCRE is the currently active context, then access is allowed. Context is switched to the object owner's context.
- Otherwise, access is denied.

### 6.2.8.6 Accessing Shareable Interface Methods

**Bycodes:**

`invokespecial`

- If the object is owned by the currently active context, then access is allowed.
- Otherwise, if the object's class implements a shareable interface, and if the interface being invoked extends the shareable interface, then access is allowed. Context is switched to the object owner's context.
- Otherwise, if the JCRE is the currently active context, then access is allowed. Context is switched to the object owner's context.
- Otherwise, access is denied.

### 6.2.8.7 Throwing Exception Objects

By code:

throws

- If the object is owned by the currently active context, then access is allowed.
- Otherwise, if the object is designated a JCRE Early Point Object, then access is allowed.
- Otherwise, if the JCRE is the currently active context, then access is allowed.
- Otherwise, access is denied.

### 6.2.8.8 Accessing Class Instance Objects

By code:

checkcast, instanceof

- If the object is owned by the currently active context, then access is allowed.
- Otherwise, if the JCRE is the currently active context, then access is allowed.
- Otherwise, if the object is designated a JCRE Early Point Object, then access is allowed.
- Otherwise, if the JCRE is the currently active context, then access is allowed.
- Otherwise, access is denied.

### 6.2.8.9 Accessing Standard Interfaces

By code:

checkcast, instanceof

- If the object is owned by the currently active context, then access is allowed.
- Otherwise, if the JCRE is the currently active context, then access is allowed.
- Otherwise, access is denied.

### 6.2.8.10 Accessing Shareable Interfaces

By code:

checkcast, instanceof

- If the object is owned by the currently active context, then access is allowed.
- Otherwise, if the object's class implements a shareable interface, and if the object is being cast into (checkcast) or is an instance of (instanceof) an interface that extends the shareable interface, then access is allowed.
- Otherwise, if the JCRE is the currently active context, then access is allowed.
- Otherwise, access is denied.

## 6.3 Transient Objects and Applet contexts

Transient objects of CLEAR\_ON\_RESET type behave like persistent objects in that they can be accessed only when the currently active applet context is the owner of the object (the currently active applet context at the time when the object was created).

Transient objects of CLEAR\_ON\_DESTROY\_TYPE type can only be created or accessed when the currently active applet context is the currently selected applet context. If any of the makeTransient() factory methods is called to create a CLEAR\_ON\_DESTROY\_TYPE transient object when the currently active applet context is not the currently selected applet context, the method shall throw a *SystemException* with reason code *ILLEGAL\_TRANSIENT*. If an attempt is made to access a transient object of CLEAR\_ON\_DESTROY\_TYPE when the currently active applet context is not the currently selected applet context, the JCRE shall throw a *SecurityException*.

Applets that are part of the same package share the same group context. Every applet instance from a package of shared all its object instances with all other instances from the same package. (This includes transient objects of both CLEAR\_ON\_DESTROY\_TYPE and CLEAR\_ON\_DESTROY\_TYPE owned by those applet instances.)

The transient objects of CLEAR\_ON\_DESTROY\_TYPE type owned by any applet instance within the same package shall be accessible when any of the applet instances in this package is the currently selected applet.

power is conditionally updated. The field or array component appears to be updated—reading the *field/array* component back yields its latest conditional value—but the update is not yet committed.

When the applet calls `jcSystem.createTransaction`, all conditional updates are committed to persistent storage. If power is lost or if some other system failure occurs prior to the completion of `jcSystem.createTransaction`, all conditionally updated fields or array components are restored to their previous values. If the applet executes an internal problem or decides to cancel the transaction, it can programmatically undo conditional updates by calling `jcSystem.abortTransaction`.

## 7. Transactions and Atomicity

### 7.1 Atomicity

Atomicity defines how the card handles the contents of persistent storage after a `select`, `failure`, or `quit` exception during an update of a single object or class field or array component. If power is lost during the update, the applet developer shall be able to rely on what the field or array component contains when power is restored.

The Java Card platform guarantees that any update to a single persistent object or class field will be atomic. In addition, the Java Card platform provides single-component-level atomicity for persistent arrays. That is, if the smart card loses power during the update of a data element (field) in an object/class or component of an array, that element will be restored to its previous value.

Some methods also guarantee atomicity for block updates of multiple data elements. For example, the `atomicity` of the `util.arraycopy` method guarantees that either all bytes are correctly copied or else the destination array is restored to its previous byte values.

An applet might not require atomicity for array updates. The `util.arraycopy` method is provided for this purpose. It does not use the transaction commit buffer even when called within a transaction in progress.

### 7.2 Transactions

An applet might need to atomically update several different fields or array components in several different objects. Either all updates take place correctly and atomically, or else all fields/components are restored to their previous values.

The Java Card platform supports a transactional model in which an applet can `begin` (the beginning of an atomic set of updates with a call to the `jcSystem.beginTransaction` method). Each object update after this

### 7.3 Transaction Duration

A transaction always ends when the JCRE receives a `commit` or `atomicUpdate` return from the applet's `select`, `createObject`, `getRecord`, `loadRecord`, `readRecord`, or `writeRecord` methods. This is true whether a transaction ends normally, with an applet's call to `commitTransaction`, or with an abortion of the transaction (either programmatically by the applet, or by default by the JCRE). For more details on transaction abortion, refer to paragraph 7.6.

*Transaction duration* is the life of a transaction between the call to `jcSystem.beginTransaction` and either:

- `call to commitTransaction or an abortion of the transaction.`

### 7.4 Nested Transactions

The model currently assumes that nested transactions are not possible. There can be only one transaction in progress at a time. If `select`, `beginTransaction`, or `commitTransaction` is called while a transaction is already in progress, a `transactionConflict` exception is thrown.

The `jcSystem.beginTransaction` method is provided to allow you to determine if a transaction is in progress.

### 7.5 Tear or Reset Transaction Failure

If power is lost (tear) or the card is reset or some other system failure occurs while a transaction is in progress, then the JCRE shall restore to their previous values all fields and array components conditionally updated since the previous call to `jcSystem.beginTransaction`.

This action is performed automatically by the JCRE when it reinitializes the card after recovering from the power loss, reset, or failure. The JCRE determines which of those objects (if any) were conditionally updated, and restores them.

**Note —** Object `power` used by instances created during the transaction that failed due to power loss or card reset can be recovered by the JCRE.

## 7.6 Aborting a Transaction

Transactions can be aborted either by an applet or by the JCRC.

### 7.6.1 Programmatic Abortion

If an applet encounters an internal problem or decides to cancel the transaction, it can programmatically undo conditional updates by calling `JCRESystem.abortTransaction()`. If this method is called, all conditionally updated fields and array components since the previous call to `activate()`, `begin()`, or `commit()` are restored to their previous values, and the JCRC is set to 0.

### 7.6.2 Abortion by the JCRC

If an applet returns from the `select()`, `delete()`, `process()`, or `install()` methods with a transaction in progress, the JCRC automatically aborts the transaction. If a return from any of `select()`, `delete()`, `process()` or `install()` methods occurs with a transaction in progress, the JCRC acts as if an exception was thrown.

### 7.6.3 Cleanup Responsibilities of the JCRC

Object instances created during the transaction that is being aborted can be deleted only if all references to these objects can be located and converted into null. The JCRC shall ensure that references to objects created during the aborted transaction are equivalent to a null reference.

## 7.7 Transient Objects

Only updates to persistent objects participate in the transaction. Updates to transient objects are never undone, regardless of whether or not they were "inside a transaction."

## 7.8 Commit Capacity

Since platform resources are limited, the number of bytes of conditionally updated data that can be accumulated during a transaction is limited. The Java Card technology provides methods to determine how much commit capacity is available on the implementation. The commit capacity represents an upper bound on the number of conditional byte updates available. The actual number of conditional byte updates available may be lower due to memory overhead.

An exception is thrown if the commit capacity is exceeded during a transaction.

## 8. API Topics

The topics in this chapter concretize the requirements specified in the *Java Card 2.1 API Draft 2 Specification*. The first topic is the Java Card I/O functionality, which is implemented entirely in the APDU class. The second topic is the API supporting Java Card security and cryptography. This section on class encapsulates the API vendor level.

### Transactions within the API

Unless specifically called out in the *Java Card 2.1 API Specification*, the implementation of the API classes shall not initiate or otherwise alter the state of a transaction if one is in progress.

### Resource Use within the API

Unless specifically called out in the *Java Card 2.1 API Specification*, the implementation shall support the invocation of API instance methods, even when the owner of the object instance is not the currently selected applet. In other words, unless specifically called out, the implementation shall not use resources such as transmit objects of **CLEAR**, **ON\_DESELECT** type.

### Exceptions thrown by API classes

All exception objects thrown by the API implementation shall be temporary ACME Entry Point Objects. Temporary ACME Entry Point Objects cannot be stored in class variables, instance variables or array components. (See 6.2.)

### ISO 7816-4 CASE 2

#### Transactions within the API

Unless specifically called out in the *Java Card 2.1 API Specification*, the implementation shall support the invocation of API instance methods, even when the owner of the object instance is not the currently selected applet. In other words, unless specifically called out, the implementation shall not use resources such as transmit objects of **CLEAR**, **ON\_DESELECT** type.

### ISO 7816-4 CASE 2

#### Transactions within the API

The APDU class encapsulates access to the ISO 7816-4 based I/O across the card serial line. The APDU Class is designed to be independent of the underlying I/O transport protocol.

The JCRE may support T=0 or T=1 transport protocols or both.

For compatibility with legacy CAD managers that do not support block chained mechanisms the APDU Class allows mode selection via the `setOutgoingMode` method.

### 8.1 The APDU Class

The APDU class encapsulates access to the ISO 7816-4 based I/O across the card serial line. The APDU Class is designed to be independent of the underlying I/O transport protocol.

Copyright © December 14, 1998 Sun Microsystems, Inc. 8-1

### 8.1.1 T=0 specifics for outgoing data transfers

For compatibility with legacy CAD managers that do not support block chained mechanisms the APDU Class allows mode selection via the `setOutgoingMode` method.

**8.1.1.1 Constrained transfers with no chaining**  
 When the no chaining mode of output transfer is requested by the applet by calling the `setOutgoingMode` method, the following protocol sequence shall be followed.

**Note** — when the no chaining mode is used calls to the `waitExternal` method shall throw an `APIUsageException` with reason code `ILLEGAL_USE`.

#### Notation

$l_e$  = CAD expected length.

$l_r$  = Applet response length set via `setOutgoingLength` method.

$<INS>$  = the prefix of byte equal to the incoming header INS byte, which indicates that all data bytes will be transferred next.

$<IN2>$  = the prefix of byte following the complement of the incoming header INS byte, which indicates about 1 data byte being transferred next.

$<SW1,SW2>$  = the response status bytes as in ISO 7816-4.

#### ISO 7816-4 CASE 2

$l_e == l_r$

1. The card sends  $l_e$  bytes of output data using the standard T=0  $<INS>$  or  $<IN2>$  procedure byte mechanism.

2. The CAD sends GET RESPONSE command with  $l_e == l_r$ .

3. The card sends  $l_e$  bytes of output data using the standard T=0  $<INS>$  or  $<IN2>$  procedure byte mechanism.

4. The card sends  $<SW1,SW2>$  completion status to completion of the Applet `process` method.

$l_r > l_e$

1. The card sends  $l_e$  bytes of output data using the standard T=0  $<INS>$  or  $<IN2>$  procedure byte mechanism.

2. The card sends  $<SW1, l_e - l_r>$  completion status bytes.

3. The CAD sends GET RESPONSE command with new  $l_e <= l_r$ .

4. The card sends  $<SW1,SW2>$  completion status to completion of the Applet `process` method.

## Java™ Card™ Runtime Environment (JCRE) 2.1 Specification

### Java™ Card™ Runtime Environment (JCRE) 2.1 Specification

1. Repeat steps 2-4 as necessary to send the remaining output data bytes (Lr) as required.
2. The card sends <SW1,SW2> completion status on completion of the Apdulet-process method.

#### ISO 7816-4 CASE 4

In Case 4, Lc is determined after the following initial exchange:

1. The card sends <0x61,1> serial bytes
2. The CAD sends GET RESPONSE command with Lc <= Lr.

The rest of the protocol sequence is identical to CASE 2 described above.

If the applet aborts early and sends less than Lc bytes, zeros may be sent instead to fill out the length of the transfer expected by the CAD.

#### 8.1.1.2 Regular Output transfers

When the no chaining mode of output transfer is not requested by the applet, when the setOutgoingLength method is used, the following protocol sequence shall be followed:

Any ISO-7816-14 compliant T=0 protocol transfer sequence may be used.

**Note** – The waitExtention method may be invoked by the applet between successive calls to sendBytes or sendBytelong methods. The waitExtention method shall request an additional work waiting time (ISO 7816-3) using the Dn60 procedure byte.

#### 8.1.1.3 Additional T=0 requirements

At any time, when the T=0 output transfer protocol is in use, and the APDU class is awaiting a CTR RESPONSE command from the CAD in return to a response status of <0x00, 1> from the card, if the CAD sends a different command, the card driver or the stdOutPutLong methods shall throw an APDUException with reason code SQL\_70\_GBYTESOURCE.

Call to sendBytes or sendBytelong methods from this point on shall result in an APDUException with reason code SQL\_70\_GBYTESOURCE. If an ISOException is thrown by the applet after the HQ\_70\_GETRESPONSE exception has been thrown, the JCRE shall disregard the response status in the reason code. The JCRE shall return APDU processing with the newly received command and reason APDU disqualifying.

### 8.1.2 T=1 specifics for outgoing data transfers

#### 8.1.2.1 Constrained transfers with no chaining

When the no chaining mode of output transfer is requested by the applet by calling the setOutgoingChaining method, the following protocol sequence shall be followed:

Notation

Lc = CAD expected length.

Lr = Applet response length set via setOutgoingLength method.

#### 8.2 The security and crypto packages

The getInstance method in the following classes return an implementation instance in the context of the calling applet of the requested algorithm:

javacard.security.MessageDigest  
javacard.security.Signature  
javacard.security.RandomData  
javacard.crypto.Cipher.

An implementation of the JCRE may implement 0 or more of the algorithms listed in the API. When an algorithm that is not implemented is requested this method shall throw a CryptoException with reason code SQL SUCH\_ALGORITHM.

Implementations of the above classes shall extend the corresponding base class and implement all the abstract methods. All data allocation associated with the implementation instance shall be performed at the time of instance construction to ensure that any lack of required resources can be flagged early during the installation of the applet.

Similarly, the buildKey method of the javacard.security.KeyBuilder class returned on implementation of instance of the requested Key type. The JCRE may implement 0 or more types of keys. When a key type that is not implemented is requested, the method shall throw a CryptoException with reason code SQL SUCH\_ALGORITHM.

**Java Card™ Runtime Environment (JCRE) 2.1 Specification**

Implementations of Key types shall implement the associated interface. All data allocation associated with the key implementation instance shall be performed at the time of instance construction to ensure that any lack of required resources can be flagged early during the instantiation of the applet.

---

### 8.3 JCSystem Class

In Java Card 2.1, the getVersion method shall return (short) 0x0201.

## 9. Virtual Machine Topics

The topics in this chapter detail virtual machine specifics.

### 9.1 Resource Failures

A lack of resources condition (such as heap space) which is recoverable shall result in a `SystemException` with reason code 10, `TRANSIENT_SPACE` to indicate lack of transient space.

All other (non-recoverable) virtual machine errors such as stack overflow shall result in a virtual machine error. These conditions shall cause the virtual machine to fail. When such a non-recoverable virtual machine error occurs, an implementation can optionally require the card to be muted or blocked from further use.

Obviously, a JCРЕ implementer could choose to implement the Installer as an Applet. If so, then the Installer might be coded to extend the Applet class and expand to invocations of the select, process, and deselect methods.

If a JCРЕ implementer could also implement the Installer in other ways, as long as it provides the SELECTable behavior to the outside world. In this case, the JCРЕ implementer has the freedom to provide some other mechanism by which APDUs are delivered to the Installer code module.

## 10. Applet Installer

Applet installation on smart cards using Java Card technology is a complex topic. The Java Card API 2.1 is intended to give JCРЕ implementers as much freedom as possible in their implementations. However, more basic common specifications are required in order to allow Java Card applets to be installed without knowing the implementation details of a particular installer.

This specification defines the concept of an Installer and specifies minimal installation requirements in order to achieve interoperability across a wide range of possible installer implementations.

The Applet Installer is an optional part of the JCРЕ 2.1 Specification. Thus is, an implementation of the JCРЕ does not necessarily need to include a post-resume Installer. However, if implemented, the Installer is required to support the behavior specified in section 9.1.

### 10.1.2 Installer AID

Because the Installer is SELECTable, it shall have an AID. JCРЕ implementers are free to choose their own AID by which their Installer is selected. Multiple installers may be implemented.

#### 10.1.3 Installer APDUs

The Java Card API 2.1 does not specify any APDUs for the Installer. JCРЕ implementers are entirely free to choose their own APDUs commands to direct their Installer in its work.

The model is that the Installer on the card is driven by an installation program running on the CAD. In order for installation to succeed, this CAD installation program shall be able to:

- Recognize the card.
- SELECT the Installer on the card.
- Drive the installation process by sending the appropriate APDUs to the card Installer. These APDUs will contain:
  - Authentication information, to ensure that the installation is authorized.
  - The AID of code to be loaded into the card's memory.
  - Linkage information to link the applet code with code already on the card.
  - Instance initialization parameter data to be sent to the applet's install method.

The Java Card API 2.1 does not specify the details of the CAD installation program nor the APDUs passed between it and the Installer.

#### 10.1.4 Installer Behavior

JCРЕ implementers shall also define other behaviors of their Installer, including:

- Whether or not installation can be aborted and how this is done.
- What happens if an exception, reset, or power fail occurs during installation.
- What happens if another applet is selected before the Installer is finished with its work.
- When another applet is selected (or when the card is reset or when power is removed from the card), the Installer becomes deselected and remains deselected until the next time that it is SELECTed.

### 10.1.1 Installer Implementation

The Installer need not be implemented as an applet on the card. The requirement is only that the Installer functionally be SELECTable. The corollary to this requirement is that Installer component shall not be able to be invoked while a non-Installer applet is selected nor when no applet is selected.

## 10.1.5 Installer Privileges

Although an installer may be implemented as an applet, an installer will typically require access to features that are not available to "other" applets. For example, depending on the JCRE implementation's implementation, the installer will need to:

- Read and write directly to memory, bypassing the object system and **standard security**.
- Access objects owned by other applets or by the JCRE.
- Invoke non-native protocol methods of the JCRE.
- Be able to invoke the `install` method of a newly installed applet.

Again, it is up to each JCRE implementor to determine the installer implementation and supply such features in their JCRE implementations as necessary to support their installer. JCRE implementors are also responsible for the security of such features, so that they are not available to normal applets.

## 10.2 The Newly Installed Applet

There is a single interface between the installer and the applet that is being installed. After the installer has correctly prepared the applet for execution (performed steps such as loading and linking), the installer shall invoke the `applet.install()` method. This method is defined in the `Applet` class.

The precise mechanism by which an applet's `install` method is invoked from the installer is a JCRE implementation-defined implementation detail. However, there shall be a control switch so that any context-related operations performed by the `install` method (such as creating new objects) are done in the context of the new applet and not in the context of the installer. The installer shall also ensure that array objects created during applet class initialization (`Client`) methods are also owned by the context of the new applet.

The installation of an applet is deemed complete if all steps are completed without failure or an exception being thrown, up to and including successful return from executing the `Applet.register` method. At that point, the installed applet will be selectable.

The maximum size of the parameter data is 32 bytes. And for security reasons, the `param` parameter is saved after the return (just as the APPU buffer is forced on return from an applet's `process` method.)

### 10.2.1 Installation Parameters

Other than the maximum size of 32 bytes, the Java Card API 2.1 does not specify anything about the contents of the installation parameter byte array argument. This is fully defined by the applet designer and can be in any format desired. In addition, these installation parameters are intended to be opaque to the installer.

JCRE implementors should design their installers so that it is possible for an installation program running in a CAD to specify an arbitrary byte array to be delivered to the installer. The installer simply forwards this byte array to the target applet's `install` method in the `param` parameter. A typical implementation might define a JCRE implementation-proprietary APPU command that has the semantics "call the applet's `install` method passing the contents of the accompanying byte array."

```

public final static byte OFFSET_AC = 4;
public final static byte OFFSET_CTRNA = 5;
public final static byte CTR_AC_1507016 = 0x00;
public final static byte INS_SELECT = (byte) 0x01;
public final static byte INS_EXTERNAL_AUTHORITY = (byte) 0x01;

Class javaxacm.framework.PNException
public static final short ILLEGAL_VALUE = 1;

Class javaxacm.framework.SystemException
public static final short ILLEGAL_VALUE = 0;
public static final byte CLEAR_ON_RESET = 1;
public static final byte CLEAR_ON_PULSE = 2;

Class javaxacm.security.CryptoException
public static final short ILLEGAL_VALUE = 1;
public static final short NO_SUCH_ALGORITHM = 2;
public static final short ILLEGAL_TRANSIENT = 3;
public static final short NO_RESPONSE = 5;

Class javaxacm.security.KeyBuilder
public static final byte TYPE_RSA_PRIVATE = 1;
public static final byte TYPE_DSS_TRANSIENT_KEY = 2;
public static final byte TYPE_DSA_PUBLIC = 4;
public static final byte TYPE_DSA_PRIVATE = 5;
public static final byte TYPE_DSA_CRT_PRIVATE = 6;
public static final byte TYPE_DSA_PUBLIC = 7;
public static final byte TYPE_DSA_PRIVATE = 8;
public static final short LENGTH_DSS = 64;
public static final short LENGTH_DSAKEY = 128;
public static final short LENGTH_DSA = 192;
public static final short LENGTH_RSA_512 = 512;
public static final short LENGTH_RSA_768 = 768;
public static final short LENGTH_RSA_1024 = 1024;
public static final short LENGTH_RSA_2048 = 2048;
public static final short LENGTH_DSA_512 = 512;
public static final short LENGTH_DSA_1024 = 1024;
public static final short LENGTH_DSA_2048 = 2048;

Class javaxacm.security.MessageDigest
public static final byte ALG_SHA = 1;
public static final byte ALG_MD5 = 2;
public static final byte ALG_RIPEMD64 = 3;

Class javaxacm.security.RandomData
public static final byte ALG_PSEUDO_RANDOM = 1;
public static final byte ALG_SECURE_RANDOM = 2;

Class javaxacm.security.Signature
public static final byte ALG_DSS_MDC4_HDPA = 1;
public static final byte ALG_DSS_MDS = 2;
public static final byte ALG_DSS_MDC4_TS977101 = 3;

```

Java™ Card™ Runtime Environment (JCRE) 2.1 Specification

```
public static final byte AAC_DES_MAC_1502797_M1 = 41;
public static final byte AAC_DES_MAC_1502797_M2 = 51;
public static final byte AAC_DES_MAC_1502797_M2 = 61;
public static final byte AAC_DES_MAC_GRC15 = 71;
public static final byte AAC_RSA_HA_J009796 = 91;
public static final byte AAC_RSA_HA_PRC51 = 101;
public static final byte AAC_RSA_HA_PRC51 = 111;
public static final byte AAC_RSA_RSAPD160_RS09796 = 121;
public static final byte AAC_RSA_RSAPD160_RS09796 = 131;
public static final byte AAC_RSA_RSAPD160_RS09796 = 141;
public static final byte MODSIGN = 11;
public static final byte REDVERIFY = 31;

Class|javacard.crypto.Cipher
public static final byte AAC_DES_MAC_NOPAD = 1;
public static final byte AAC_DES_MAC_RS09793_M1 = 21;
public static final byte AAC_DES_MAC_RS09793_M2 = 31;
public static final byte AAC_DES_MAC_RS09793_M2 = 41;
public static final byte AAC_DES_MAC_RS09793_M2 = 51;
public static final byte AAC_DES_MAC_RS09793_M2 = 61;
public static final byte AAC_DES_MAC_RS09793_M2 = 71;
public static final byte AAC_DES_MAC_RS09793_M2 = 81;
public static final byte AAC_DES_MAC_RS09793_M2 = 91;
public static final byte AAC_RSA_PRC51 = 101;
public static final byte AAC_RSA_PRC51 = 111;
public static final byte AAC_RSA_PRC51 = 121;
```

control is changed to correspond to the applet context that owns the object. When that method returns, the previous context is restored. Invocations of static methods have no effect on the currently active context. The currently active context and sharing status of an object together determine if access to an object is permissible.

**Currently selected applet.** This ICRU keeps track of the currently selected Java Card applet. Upon receiving a SELECT command with this applet's AID, the JCRE makes this applet the currently selected applet. The JCRE sends all APDU commands to the currently selected applet.

**REPROD** is an acronym for **R**emotely **P**rogrammable **R**eadOnly **D**evice.

**Firewall** (see **Applet Firewall**).

**Framework** is the set of classes that implement the API. This includes core and extension packages. Responsibilities include dispatching of APDUs, applet selection, managing atomicity, and launching applets.

**Garbage collection** is the process by which dynamically allocated memory is automatically reclaimed during the execution of a program.

**Instance variables**, also known as fields, represent a portion of an object's instant state. Each object has its own set of instance variables. Objects of the same class will have the same instance variables, but each object can have different values.

**Implementation**, in object-oriented programming, means to produce a particular object from its class template. This involves allocation of a data structure with the types specified by the template, and initialization of instance variables with either default values or those provided by the class's constructor function.

**JAR** is an acronym for Java Archive. JAR is a platform-independent file format that combines many files into one.

**Java Card Runtime Environment (JCRE)** consists of the Java Card Virtual Machine, the framework, and the associated native methods.

**JC2141** is an acronym for the Java Card 2.1 Reference Implementation.

**JCRE implementer** refers to a person creating a vendor-specific implementation using the Java Card API. JCVM is an acronym for the Java Card Virtual Machine. The JCVM is the foundation of the CP card architecture. The JCVM executes byte code and manages classes and objects. It enforces separation between applications (firewalls) and enables secure data sharing.

**JDK** is an acronym for Java Development Kit. The JDK is a Sun Microsystems, Inc. product that provides the environment required for programming in Java. The JDK is available for a variety of platforms, but most notably Sun Solaris and Microsoft Windows.

**Method** is the name given to a procedure or routine, associated with one or more classes, in object-oriented languages.

**Namespace** is a set of names in which all names are unique.

**Object-Oriented** is a programming methodology based on the concept of an object, which is a data structure encapsulated with a set of routines, called methods, which operate on the data.

**Objects**, in object-oriented programming, are unique instances of a data structure defined according to the template provided by its class. Each object has its own values for the variables belonging to its class and can respond to the messages (methods) defined by its class.

**Currently active context.** The ICRU keeps track of the currently active Java Card applet context. When a virtual method is invoked on an object, and a context switch is required and permitted, the currently active context (see Applet execution context.)

Copyright © December 14, 1998 Sun Microsystems, Inc.

## *Java™ CAD™ Runtime Environment (JCERB) 2.1 Specification*

**Package** is a namespace within the Java programming language and can have classes and interfaces. A package is the smallest unit within the Java programming language.

**Persistent object** Persistent objects and their values persist from one CAD session to the next, indefinitely. Objects are persistent by default. Persistent object values are updated atomically using transactions. The term persistent does not mean there is an object-oriented database on the card or that objects are serialized/deserialized, just that the objects are not lost when the card loses power.

**Shareable interface** Defines a set of shared interface methods. These interface methods can be invoked from one application context when the object implementing them is owned by another application context.

**Shareable interface object (SIO)** An object that implements the shareable interface.

**Transaction** is an atomic operation in which the developer defines the extent of the operation by indicating in the program code the beginning and end of the transaction.

**Transient object** The values of transient objects do not persist from one CAD session to the next, and are stored in a default state at specified intervals. Updates to the values of transient objects are not atomic and are not affected by transactions.

1/5/99 12:49 PM Havnor:Stuff:JCRE D2 14DEC98:READ-ME-JCRE21-DF2.txt

Page 1

Date: 16 December 1998

Dear Java Card Licensee,

JCRE21-DF2-14DEC98.zip contains a second draft of the Java Card 2.1 Runtime Environment specification, dated December 14, 1998, for Licensee review and comment. We have worked to incorporate and clarify the document based upon the review feedback we've received to date.

Complete contents of the zip archive are as follows:

READ-ME-JCRE21-DF2.txt	- This READ ME text file
JCRE21-DF2.pdf	- "Java Card Runtime Environment (JCRE) 2.1 Specification" in PDF format
JCRE21-DF2-changebar.pdf	- The revised document with change bars from the previous version for ease of review.

Summary of changes:

1. This is now a draft 2 release and will be published on the public web site shortly.
2. New description of temporary JCRE Entry Point Objects has been introduced for purposes of restricting unauthorized access. Firewall chapter 6.2.1.
3. Global arrays now have added security related restrictions similar to temporary JCRE Entry Point objects. Firewall chapter 6.2.2.
4. Detailed descriptions of the bytecodes with respect to storing restrictions for temporary JCRE Entry Point Objects and Global arrays added. Chapter 6.2.8.
5. General statement about JCRE owned exception objects added in chapter 8.
6. Corrected description of Virtual machine resource failures in transient factory methods. Chapter 9.1.

The "Java Card Runtime Environment 2.1 Specification" specifies the minimum behavior and runtime environment for a complete Java Card 2.1 implementation, as referred to by the Java Card API 2.1 and Java Card 2.1 Virtual Machine Specification documents. This specification is required to ensure compatible operation of Java Card applets. The purpose of this specification document is to bring all the JCRE elements together in a concise manner as part of the Java Card 2.1 specification suite.

Please send review comments to <javaoem-javacard@sun.com> or to my address as below. On behalf of the Java Card team, I look forward to hearing from you.

Best,  
Godfrey DiGiorgi

Godfrey DiGiorgi - godfrey.digiorgi@eng.sun.com  
OEM Licensee Engineering  
Sun Microsystems / Java Software  
+1 408 343-1506 - FAX +1 408 517-5460